STUDENT NAME: _____ **STUDENT ID:** _____

CS 222/122C – Fall 2018, Final Exam

Principles of Data Management
Department of Computer Science, UC Irvine
Prof. Chen Li
(Max. Points: 100 + 10)

Instructions:

- **This exam has eight (8) questions including two extra-credit sub-questions.**
- **This exam is closed book. However, you can use one cheat sheet (A4 size).**
- **The total time is 120 minutes. So budget your time accordingly.**
- **Be sure to answer each part of each question after reading the whole question carefully.**
- **If you don't understand something, ask for clajrification.**
- **If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.**
- **Try to avoid writing your answers on the back of papers.**

QUESTION	POINTS	SCORE
1	12	
2	14	
3	12	
4	16	
5	12	
6	10	
7	20	
8	4 + 10 (with extra credit)	
TOTAL	100 + 10	

Question 1: Short questions (12 points)

a) (3 pts) In the external sort algorithm covered in lectures, during each merge phase, suppose we have ***k*** input runs with ***F*** records in each run, and they are merged into a bigger run. What is the data structure used to do the merge? What's the time complexity of merging these records? Briefly explain why.

Priority queue. log(k) * k * F * 2, since each record needs to be pushed and popped, and each operation is log(k).

b) (3 pts) As discussed in class, sort and hash are two dual methods widely used in database engines. We described three cases where these two methods were "competing," namely indexing, duplicate elimination, and equi-join. For each case, mention the winner and briefly explain why.

Indexing: B+ tree (sort) is better than a hash table, since the former can support range queries, and the complexity is very low. Many systems chose to implement B+ tree only.

Duplicate elimination: the sort-based method is better since it's less sensitive to bad data distributions. Plus the output is sorted.

Join: the hash-based join method is typically better since it has a lower memory requirement and can use additional memory more efficiently by reducing the number of disk IOs (using Simple Hash Join or Hybrid Hash Join).

c) (3 pts) Use a diagram to briefly explain how hybrid hash join combines the ideas in grace hash join and simple hash join.

Keep one big partition in memory so that its records don't need to go to the disk (same idea as simple hash join). Those "unlucky" records that need to go to disk are partitioned (same idea as grace hash join).

d) (3 pts) Use a join query to illustrate that in the buffer manager of a database, the least-recently-used (LRU) replacement policy may not be good, while the most-recently-used (MRU) replacement policy can be better. Briefly explain why.

Consider R JOIN S using block-based NLJ. Relation S has 101 pages, and the buffer pool has 101 pages, where one page is used to hold a page of R. In this case LRU will cause each read of an S page to cause a cache miss, and MRU is preferred.

Question 2: External Sort (14 points)

Suppose you just inserted records into a heap file and want to sort them using external sort. Assume:

- The file has 4,500 records.
- Each sort key is 4 bytes.
- Each rid is 8 bytes.
- Each page id is 4 bytes.
- Each record is stored as an array of 48 bytes.
- Each page size is 512 bytes.
- We use a directory-based page format as implemented in our projects.
- Each page uses 2 bytes for free-space information and 2 bytes for the number of slots.
- Each directory slot uses 4 bytes (total) for the offset and length of a record.
- There are 4 buffer pages in memory.
- In pass 0, each iteration reads 4 pages of records to generate an initial run.

For each of the following questions, make sure to write a formula before plugging in numbers in order to get partial credits in case your calculation is wrong.

a) (4 pts) Draw the page format and calculate the maximum number of records per page.

Number of records per page: floor((512 - 2 - 2) / (48 + 4)) = 9

b) (3 pts) Assume we use the same page format for sorted runs. How many sorted runs will be generated after pass 0 of external sort, and how many pages are in each generated run?

N = 4500 / 9 = 500
B = 4
#RUN = ceil(N / B) = ceil(500 / 4) = 125
#PAGE in each RUN = 4

c) (2 pts) How many passes (including pass 0) are required to sort this file?

#PASS = ceil($\log_{B-1}$(#RUN)) + 1 = ceil($\log_3$(125)) + 1 = 6

d) (2 pts) What is the total I/O cost for sorting this file, **excluding the last output I/Os**?

#I/O = N * (2 * #PASS - 1) = 500 * (2 * 6 - 1) = 5,500

e) (3 pts) What is the largest file, **in terms of the number of pages**, you can sort with just 4 buffer pages in two passes (including pass 0)? How would your answer change if you had 100 buffer pages?

N <= B * (B - 1) = 4 * 3 = 12
N <= B * (B - 1) = 100 * 99 = 9900

Question 3: Projection (12 points)
Consider a table:

`Order(orderID, storeID, numOfItems, totalPrice)`

and the following SQL query:

```
SELECT DISTINCT storeID, numOfItems FROM Order;
```

Assume:

- The `orderID` attribute is unique.
- The values of `orderID`, `storeID`, `numOfItems`, and `totalPrice` are 4 bytes each.
- The table has 300 pages.
- There are 30 buffer pages in memory.

a) (4 pts) Consider the sort-based algorithm covered in lectures. Assume when generating a run in pass 0, in each iteration we read exactly 30 pages of data to fill in the memory. How many sorted runs are produced after this pass? What is the size of each run in terms of number of pages? What is the I/O cost of this pass 0?

# of sorted runs = 300 / 30 = 10.
We remove unwanted fields in this run so the size of records will be reduced by half.
Each run hass 15 pages.
I/O cost for this pass: 300 + 150 = 450.

b) (4 pts) How many **additional** merge passes are required to compute the final result of the projection query? What is the I/O cost of these additional passes? In the calculation, ignore the output I/Os of the last pass.

We need 1 more pass to compute the final result.
I/O cost will be 150 since we have 10 runs, each of which has 15 pages.

c) (4 pts) Consider the following SQL queries on the table above:

- **`Q1: SELECT DISTINCT orderID, storeID FROM Order;`**
- **`Q2: SELECT DISTINCT storeID, totalPrice FROM Order;`**
- **`Q3: SELECT DISTINCT numOfItems, totalPrice FROM Order;`**

Which of them has the optimal physical plan with a smallest number of disk I/Os? Ignore the output I/Os of the last pass. Briefly explain your answer.

Q1 will have the optimal physical plan since OrderID is unique and there is no duplicate elimination phase. The cost for this plan will be 300 I/Os.

Question 4: Join (16 points)

Consider the following tables:

- `Flight(airline, fnumber, origin, destination)`
- `Passenger(pid, pname, airline, fnumber)`

Assume:

- The `Flight` table has a composite clustered primary index on `(airline, fnumber)`.
- `Passenger` has a clustered primary index on pid.
- Each passenger is assigned to **one and only one** flight.
- The `Flight` table has 20 pages with 50 records per page.
- The `Passenger` table has 300 pages with 100 records per page.
- We have 12 buffer pages in memory.

Consider the following query:

```
SELECT *
FROM Flight F, Passenger P
WHERE F.airline = P.airline AND F.fnumber = P.fnumber;
```

In the following calculations, ignore the IO cost of the output results.

a) (4 pts) How many disk I/Os are needed to perform page-based nested loop join? Choose the join order with fewer I/Os.

Cost: scan of outer table + (#blocks * scan of inner table)
20 + (20 * 300) = 6020.

b) (4 pts) How many disk I/Os are needed to perform block-based nested loop join?. Choose the join order with fewer I/Os.

Cost: Scan of outer table + #outer blocks * scan of inner table
20 + (20/10 * 300) = 620.

c) (4 pts) How many disk I/Os are needed in order to perform index-based nested loop join? Assume all the non-leaf nodes of indexes are cached in memory.

Cost: Scan of outer table + ((#records in outer table) * cost of finding matching tuples)
300 + (30,000 * 1) = 30,300 I/Os
Another acceptable solution is:
00 + (30,000 * (1 + 1)) = 60,300 I/Os

d) (4 pts) In class we discussed different join approaches, including block-based nested loop join, index-based loop join, sort-based join, and hash-based join. For the following query, indicate which of these approaches can achieve the best performance and briefly explain why.

```
SELECT *
```

```
FROM Flight F, Passenger P
WHERE F.airline = P.airline AND F.fnumber < P.fnumber;
```

Although we will be able to answer this query using index-based nested loop join and sort-based join, Block-based nested loop join will perform better than others in terms of disk I/Os.

Question 5: Hash Join (12 points)

Consider the following tables:

- `Celebrity(cname, birthdate, occupation)`
- `Fan(fname, birthdate, gender, cname)`

Assume:

- Each underlined attribute is unique.
- Each fan has **one and only one** celebrity.
- The `Celebrity` table has 400 pages. Each page has 50 records.
- The `Fan` table has 900 pages. Each page has 10 records.
- All celebrities have the same number of fans.

In the following calculations, ignore the IO cost of the output results.

a) (4 pts) Consider **grace hash join** with **26** buffer pages.

i) Suppose we use the join order that minimizes the I/O number. How many phases are needed? How many disk IOs? Briefly explain your answer.

Since sqrt(400) = 20 <= 24, we can perform grace hash join in just two passes.
Cost: 3 * (400 + 900) = 3,900 I/Os

ii) What is the number of partitions for `Celebrity` in the initial partitioning/build phase and what is the number of pages in each partition?

Number of partitions: 25.
Each partition has 400 / 25 = 16 pages.

b) (4 pts) Consider the **simple hash join** with **203** buffer pages. Write down the total I/O cost and briefly explain why.

We can have half of the records present in the memory using B-3 = 200 buffers. We can perform simple hash join in 2 passes, and we will only load half of the records in the second pass.
Cost: 400 + 900 + 200 + 450 + 200 + 450 = 2,600 I/Os

c) (4 pts) Consider the **hybrid hash join** with **152** buffer pages. In the partitioning/build phase, we use 1 page as the input buffer, 1 page as the output buffer for the final join results, 100 pages for the in-memory partition of the outer table, and the remaining 50 pages for the output buffer pages to do partitioning. Write down the total I/O cost and explain why.

We will use 100 pages to build an in-memory hash table, which is ¼ of pages of Celebrity. Since all celebrities have the same number of fans, the same will happen for the Fans table.
In the next phase we will use grace hash join for each partition spilled to disk.

Cost: (400 + 900) + (300 + 675) + (300 + 675) = 3,250 I/Os

Question 6: Cost Estimation (10 points)
a) (4 pts) Consider the table

`Employee(name, birthDate, salary)`

We want to build histograms on attributes, namely equi-width histogram and equi-height histogram. For the following two scenarios, indicate which type of histogram is more desirable and explain why.

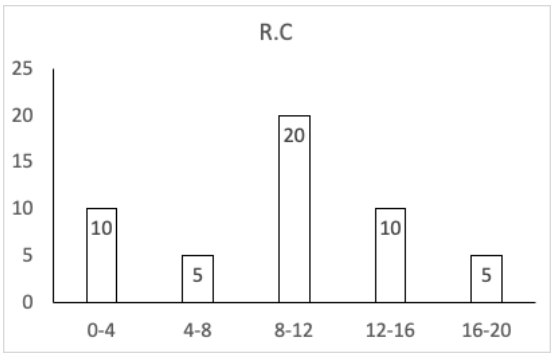
(i) We want to build a histogram on the `birthDate` attribute, and no index on this attribute is available. Assume people's birthdays are evenly distributed.

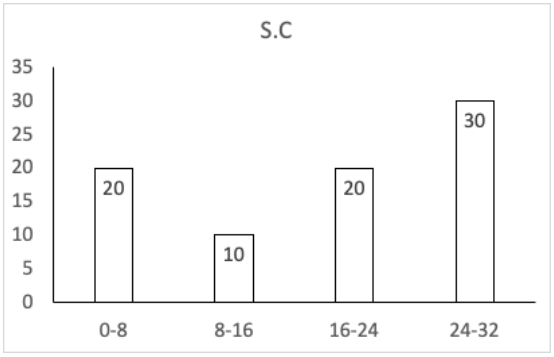
Equi-width is better since data is not skewed.

(ii) We want to build a histogram on the `salary` attribute, and we have a B+ tree index on this attribute. Assume salaries are skewed into a small range.

Equi-height is better since data is skewed and the attribute values are already sorted in the B+ tree.

b) (6 pts) Suppose we have two relations `R(A, C)` and `S(B, C)`, where `R.A` and `S.B` are unique. We have an equi-width histogram on `R.C` and `S.C` respectively (as shown below), and the values of `R.C` are a subset of the values of `S.C`. Describe a method to use these histograms to estimate the size of `R` ⋈ `S`, and give the estimated size based on the given histograms.





The basic idea is to compute the join result size for each range of R.C, and then compute the sum of these values. Since R.C is a subset of S.C, each value of R.C must have at least one match to R.C.

In this example, we consider the join result size for each range of R.C as follows:

- 0-4: 10 values for R.C and 10 values for S.C. the estimated join result size is 10;
- 4-8: 5 values for R.C and 10 values for S.C. the estimated join result size is 10 (not 5, because R.C is a subset of S.C);
- 8-12: 20 values for R.C and 5 values for S.C. the estimated join result size is 20;
- 12-16: 10 values for R.C and 5 values for S.C. the estimated join result size is 10;
- 16-20: 5 values for R.C and 10 values for S.C. the estimated join result size is 10.

The total size is: 10+10+20+10+10 = 60

Question 7: Query Optimization (20 points)

a) (4 pts) Give two reasons why the System-R query optimizer only considers left-deep join plans during optimization.

1. Reduce the search space
2. Allow more opportunities to perform pipelining (i.e., not writing intermediate results to disk).

Consider the following three tables about students and courses, and their relationships, together with available B+ tree indexes:

- `Student(sid, sname, GPA)`
  - Clustered index on "**`sid`**" (meaning student id), i.e., records are stored in the leaf nodes of the B+ tree;
  - Unclustered index on "**`sname`**";
  - Unclustered index on "**`GPA`**";
- `Course(cid, cname, year, quarter)`
  - Clustered index on "**`cid`**" (meaning course id), i.e., records are stored in the leaf nodes of the B+ tree;
  - Unclustered index on "**`year`**";
- `Takes(sid, cid, date)`

- With information about which students took which classes and when;
- Clustered index on "**sid,cid**", i.e., records are stored in leaf nodes of the B+ tree;
- Unclustered index on "**cid**".

Consider the following query

```
SELECT sid, COUNT(*)
FROM Student S, Takes T, Course C,
WHERE S.sid = T.sid AND T.cid = C.cid AND GPA>3.0 AND year = 2017
GROUP BY S.sid
```

b) (6 pts) For each of the following tables, write down all the access methods considered by the System-R optimizer. Specify which of them will be considered for the next phase and explain why. Write down all the available interesting orders for each access method.

- Takes (sid, cid, date)

1. B+ tree scan on sid,cid, interesting order on sid; kept;
2. B+ tree scan on cid, interesting order on cid; kept;
3. B+ tree search on sid for a constant, kept for a later index-based join;
4. B+ tree search on cid for a constant, kept for a later index-based join.

Note: the search condition in 3 actually only uses sid, since we would not join Course with Student together beforehand. However, if your answer uses sid+cid, we'll mark that error but not deduct points.

- Student (sid, sname, GPA)

1. B+ tree scan on sid, interesting order on sid; kept
2. B+ tree search on GPA; kept if cost is less than the access method 1
3. B+ tree search on sid for a constant, kept for a later index-based join
4. (optional: B+ tree search on sname; not kept)

c) (4 pts) Write down all the **sets** of join relations considered by the System-R query optimizer.

- (Student, Takes)
- (Takes, Course)
- (Student, Takes, Course)

d) (6 pts) A **parameterized query** is a type of query where some constant values in the WHERE conditions have been replaced by some variables. These values are only known to the query optimizer during runtime. An example is shown as follows, where the actual value of "?" is only known to the query optimizer during runtime:

```
SELECT S.sid, COUNT(*)
FROM Student S, Course C, Takes T
```

```
WHERE S.sid = T.sid AND T.cid = C.cid AND GPA > 3.0 AND year = ?
GROUP BY S.sid
```

To optimize parameterized queries, a naive approach is to always optimize it separately whenever a query is about to be executed. Suggest a more efficient method to optimize parameterized queries without re-optimizing it everytime, and use the above query example to illustrate the method. You can assume that there is a histogram built on `Course.year`. Your solution must deal with the fact that the query selectivity is not known beforehand.

The basic idea is to generate multiple query plans, and each corresponds to a range of selectivities. During runtime, we can simply estimate the selectivity and fetch the corresponding optimal plan. The key observation is that the System-R optimizer only relies on query selectivities instead of query predicates.

Question 8: Performance Robustness (4 + (10 extra-credit points))

Performance robustness is a desirable property of query operators. It states that the running time of an operator should degrade gracefully as the input size grows. For example, the filter operator is robust since its running time is linear w.r.t. to the number of input records. However, the external-sort operator as covered in class may not be robust. In particular, if the input size is slightly larger than the available sort memory, the running time could increase a lot (because of disk I/Os) compared to the case where the input totally fits into memory.

a) (4 pts) Give two reasonable arguments why performance robustness is a desirable property of a database system.

1. It makes the query performance more predictable;
2. It simplifies cost estimation and query optimization.

Note: robustness is not related to absolute disk I/Os, scalability, or efficiency. It only requires the overall running time of an operator increases **continuously.** In either case, the running time of an operator will increases with more data.

b) (Extra credit: 5 pts) Suggest a solution to improve the robustness of **pass 0** in the external sort algorithm when the input is slightly larger than the sort memory. In other words, the performance should degrade gracefully as the input becomes larger.

The key problem here is that when the memory is full, we will flush the entire memory as a sorted run. If the input data is slightly larger than the main memory, there will be a sudden jump of running time.

To address problem, we first reserve one page X (whose purpose will be clear later). Suppose there are B pages in memory. We read B-1 pages from the input and produce a sorted run. Instead of flushing the entire sorted run (called Run 1) to disk, we only flush the last page with the largest record values to disk, and use that page to read one more page from the input file (as a separate run, called Run 2). We keep flushing the tail (largest) page from Run 1 to disk and using the freed page to read one more page from the input file to Run 2, until there are no more pages in the input file. Now we have two runs, namely Run 1 (including some pages on disk) and Run 2 (in memory).

Then we merge these two runs in memory and output them to the disk using the pre-allocated page X. At the end of the merge process, we will read those on-disk pages of Run 1 one by one.

c) (Extra credit: 5 pts) Another problem with the external sort is that during the merge phase (possibly with multiple merge passes), if the number of runs to be merged is slightly larger than the main memory, then we need one more merge pass that will incur a lot of additional disk IOs. Suggest a solution to improve the robustness of external sort during the merge phase.

Suppose we can only merge 4 runs at a time, but we have 5 runs. Based on the original merge algorithm, we will need two passes, which read/write the input file two times. To address this problem, we should first merge 2 smallest runs so that we only have 4 runs. Then, we merge these 4 runs together in one pass. In this case, we only use a partial pass to read/write 2 runs, instead of reading the entire input file.

To further optimize this process, we should pre-merge the smallest runs during pass 1 so that the number of remaining passes is minimized.